AN4655
Application note

Virtually increasing the number of serial communication peripherals in STM32 applications

Introduction

Application engineers often face the problem of limited number of serial communication peripherals of a microcontroller that, on the other hand fulfills all the other application requirements thanks to its features and performance. Sometimes they obviate by switching to a higher level microcontroller with sufficient number of communication peripherals. This migration brings with it additional (often unused) performance and functionality, in most cases unneeded and not used by application, in addition to increased costs and PCB complexity.

A frequent case is when full (or specific) functionality is not required for each and every channel, in this case the communication flow and its control can be simplified radically (e.g. communication is required at specific modes or time slots only, communication speed can be lower, correct timing is not strictly required for all the signals, simplified protocol or flow is acceptable). In these specific cases the user would really benefit from methods on how to supplement the missing channel(s) with current HW, to avoid needless migrations.

On the other side, reaching nearly full compatibility with all the native HW features is hardly achievable at an alternate channel, besides costing a lot in terms of code and performance. Usually it is preferable to abandon some specific requirements with the goal of simplify the concept of the alternated peripheral channel. A crucial point here is to recognize those undemanding channels in the application, which can be altered with lower effort and loss of performance.

This application note provides a basic overview of the described issue, and will help the application engineers to identify possible alternate methods when implementing missing communication channels. It applies to all STM32 microcontrollers, as the discussed peripherals are present on all products of this class.

Additional informations and examples can be found in the following reference documents, describing actual cases and solutions:

- TN0072, Software toolchains and STM32 features;
- UM0892, STM32 ST-LINK Utility software description;
- AN4457, STM32F4 full Duplex UART emulation.

The last cited document is being developed together with other application notes on the same topic, user should check their availability on *www.st.com*.

Contents

List of tables

List of figures

1 HW methods

If addition of native HW peripherals is not acceptable (e.g. by adding external components like a dedicated communication coprocessor unit), then the limited number of peripherals can be compensated by the following basic methods:

1. Use internal remapping option of dedicated peripheral instances, to change the inputs and outputs available at different ports, thus establishing a temporary connection of the microcontroller with a different channel (the user can find informations about alternate function capability in the pin-out description of datasheets, for each product and its implemented peripherals).
2. Multiplexing communication channels into a single communication port while using multiplexing logic.
3. Using SWO feature.

The first two methods provide service on selected channel exclusively in dedicated time slots, and can be used only when contemporary operation and permanent monitoring of altered channels is not strictly required, as it happens, for example, in:

- service channels;
- occasional communication with remote devices;
- monitoring of sensors with slow data value change;
- channels used for periodical backup.

These methods are suitable for master communications when any slave activity is expected upon master request, or when some communication with slave can be missed and can be easily back re-synchronized when master reconnects back to the channel to be serviced (listened and talked to). It could be used at slave side, too, when some additional control signals are provided between slave and master to inform the slave that connection is established (e.g. chip select or channel ready/busy signalizing when slave can/can't operate with the channel).

Anyway, user should take care about inactive state at GPIOs associated with a peripheral instance bus signals when alternate function is remapped or multiplexed to another channel. A proper setting ensures emulation of idle condition and prevents any stuck at channels not currently selected or just not used for communication. This can be achieved with a correct configuration of GPIO port registers, and partially by adding external components (pull up/pull down resistors). Note that GPIO logic commonly overtakes control over the associated pins when alternate functionality is removed (remapped) from the port and user has to prevent any bus false state detection on such temporarily frozen channels.

For multiplexing, user can use specific control of I/O switches at dedicated groups of routing interface if it is available for the product. User must ensure that a proper combination (pair) of switches within a single group be closed in time to let the signal pass through the desired path. User has to dedicate additional IOs for such switching, so they are lost for other purposes, and there is additional PCB routing.

The principle is shown in *Figure 1* and *Figure 2*.

Figure 1. Remapping option principle

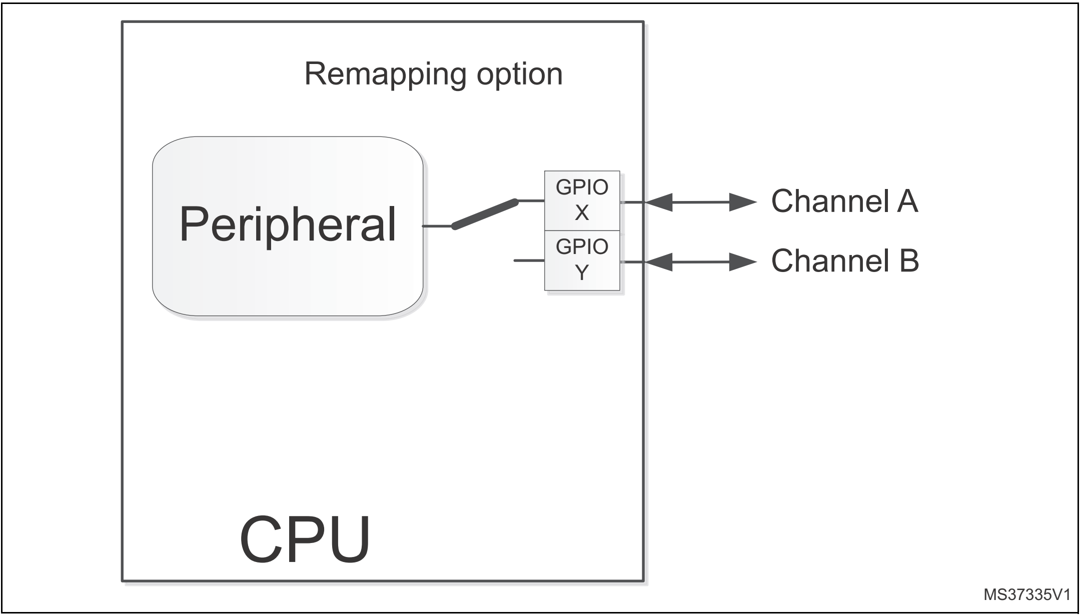
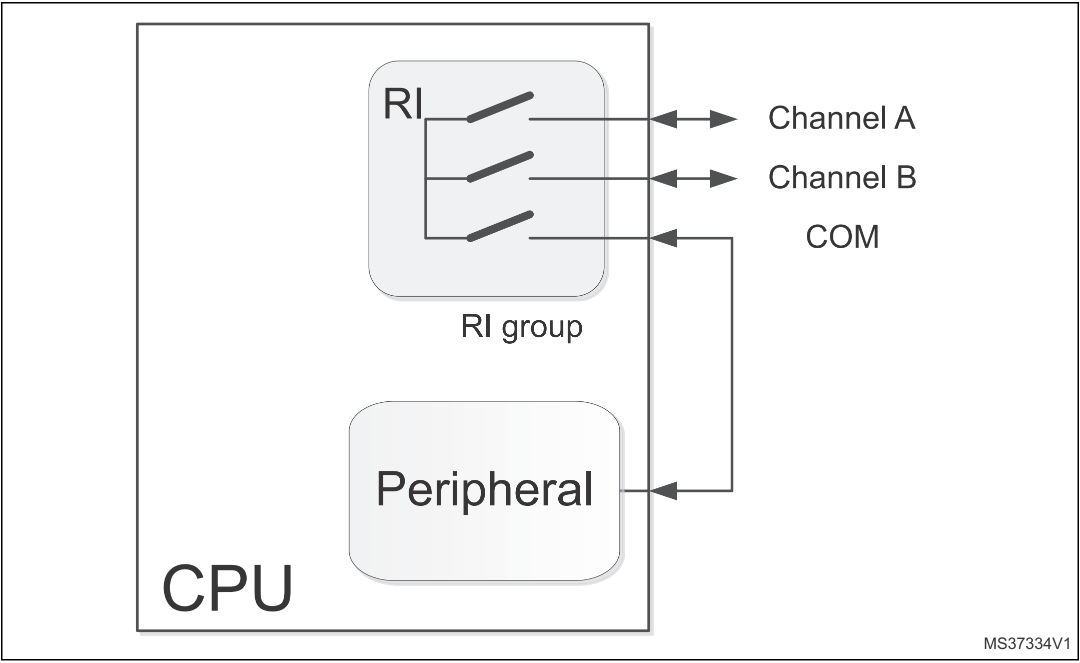


Figure 2. Multiplexing principle while using dedicated group of routing interface (RI)



During debug, an additional communication port is often needed to communicate the application specific diagnostic information and system monitoring during real run. User can setup virtual COM communication port at SWO pin while using specific instrumentation trace macro-cell (ITM) and Serial Wire Viewer (SWV) interface.

The ITM is an application-driven trace source that supports printf style debugging to trace Operating System (OS) and application events, and emits diagnostic system information via STLink tool. The ITM module is available at some latest products and so save standard communication peripherals for an application purpose. It is supported by some IDEs and ST HAL firmware. It is enough to call the *ITM_SendChar()* function inside *fputc()* to send a byte over the interface and configure the debug module by assigning TRACE pin for Asynchronous trace mode.

2 SW methods

2.1 Compatibility factors

When a developer starts to think about replacing any peripheral HW interface by emulation of its functionality by GPIO SW control, he should consider carefully the compatibility factors and the truly required levels.

2.1.1 Functionality (not related to timing)

Normally, this is controlled by configuration and status registers (e.g. data or protocol format, communication speed, parity, control of associated signals, CRC calculation etc.). Universality is crucial question here. The development can be considerably simplified when the requirements are limited to match the application needs (fixed clock phase or polarity, order and number of bits, simplex flow, no parity or CRC support etc.).

When trying to achieve wider flexibility and capability with target to use the FW in a common way, the development complexity can increase significantly, including claims for testing and validation phases. Basically, a complete state diagram layout with all transitions and its conditions, drawn in the initial planning phase, is a good starting points as it helps to understand all the requirements and the complexity of the required emulation.

2.1.2 Timing

Correct shape of signals (edges, duty cycle), inter-timing between clock and data access (e.g. data setup or hold times) and correct signal sampling (position and number of samples) has to be considered here. These requirements are mostly limited by lack of performance capability and limitations of any fast IO control when it is done by SW.

Some cores offer dedicated internal buses for IO control to perform very fast IO access and driver capability (e.g. specific IOPORT of Cortex®-M0+ core).

Very good results can be achieved when GPIO signals are provided while using output control done by HW e.g. at timers' events (e.g. output compare feature). User can use wide STM32 architecture capabilities to synchronize peripherals internally but the emulated control flow logic has to be supervised by software inside interrupt services at this case with all the negative consequences and sure timing constrain especially.

2.1.3 HW Interface

Concerning specific pin requirements (like true open drain driving, slope of edges), those dedicated to a functionality (by design) can have their native HW supporting feature(s). This means that it will be almost impossible to replicate such feature(s) when trying to emulate the same functionality at some other pins where such functionality is not supported natively

At minimum, a correct GPIO configuration (speed limitation, output drivers) should not make the case worse, and correspond as much as possible to the given purpose.

2.1.4 API interface

Application of an API interface built upon a structure of software low level drivers can bring good flexibility, but it can increase the burden on system performance, because the complexity of solution grows fast with the intended scope. User usually defines API functions to configure a dedicated peripheral, to perform single read and write operations, or to open and close communication sessions. The process events are normally handled by callbacks performing the correct flow of data or occasional error states. Universality pains here again and the task can be simplified considerably when some functionality can be limited or even removed and not supported. Big advantage of this method is better code readability while the code can remain well structured. Drawbacks are time and performance constraints.

2.2 Applicable methods

Two basic methods can be applied when emulating a communication peripheral by SW:

1. bit banging;
2. a combination of HW and SW control of GPIOs.

2.2.1 Bit banging

Signals are emulated by direct control of IO ports via SW purely. This method can be used at synchronous protocols especially (SPI, I2C) where signal timing usually isn't so critical. *SPI bit-banging sequence* and *I2C master bit-banging sequence* indicate possible solutions.

2.2.2 Combination of HW and SW control of GPIOs

Signals are emulated indirectly while using capability of specific HW to control IOs at required times. The communication logic is then implemented at software level, mostly at interrupt services driven from the internal HW events. User can study the principle in *UART duplex channel based on combination of HW and SW control of GPIOs*, demonstrating the application of this method in a real case.

The user can find another possible solution in already cited AN4457.

This document describes in detail HW synchronization between timers and DMA: events from a dedicated timer synchronize and trigger sequences of DMA transfers between a specific data buffer and GPIO registers. Content of data buffer can be either source of GPIO output control sequence stored in advance by SW in a a dedicated memory area, or a sequence of samples read and stored from GPIO input to be analyzed after the closing of the transaction. These transfers are provided at exact intervals during communication transactions, while a simultaneous flow in both direction can proceed under independent control.

2.3 Common principles of the methodology

2.3.1 Master vs. slave, clock signal aspects

Master emulation is an easier case in principle. It can determine time slot flow communication, data flow can be handled off line (e.g. CRC calculation can be prepared in advance and checked afterwards over the data package) so the physical data flow can be handled in a relatively fast way.

Master providing synchronous clock is the simplest case. Master starts the flow, handles clock & data signal, timing is not crucial here and it is nearly free except some if some specific limitation has to be respected on the (usually slower) slave side (e.g setup or hold times, communication timeouts).

Asynchronous master is much more demanding as lack of performance could make a wrong effect and corrupt the data flow. Master has to be able to provide always correct timing of data output and/or input at this case.

Emulation of a slave is usually the worst case (independently if clock signal is synchronous or not), as slave has to:

- be always ready for the transaction slot;
- be well synchronized with the slot start (especially with asynchronous clock);
- keep correct timing within the slot (derived from time calculations or clock signal change);
- handle correct data flow (input, output, or even both signals).

2.3.2 Interrupts and DMA vs. bit-banging

Interrupts derived from GPIO, timer or DMA events are easier to handle and more systematic at FW level. Required protocol can be well synchronized with data flow if system has enough performance capacity to handle all needed software procedures. User can control all the flow and service the events either directly inside the interrupt services or by dedicated callbacks provided by API. The second way gives a space to sure level of universality and keeps the SW well structured. But this way requires wider reserve of microcontroller performance to be sufficient to manage all the communication events at real time. This could be a limitation when supporting higher communication speeds while duplex data flow is handled especially.

When DMA is used, the user should consider the capacity of internal buses, especially when additional DMA processes are handled in parallel, and respect their priorities. Every DMA transfer blocks the involved buses for a few cycles and a significant latency of a DMA service can appear when the system handles interrupt service entry and return, or when specific instructions requiring bus traffic are executed.

Let us imagine a specific case when a signal edge is handled by external GPIO interrupt. The FW developer has to calculate with the:

- interrupt service latency (few CPU cycles, depends on other pending interrupts and their priorities as well);
- process entry into interrupt service (context saving delay);
- state logic (time needed for detection communication phase and action to be performed by the software);
- flow dedicated action (set data output, sample data input, handle data store, change of clock signal, setup timing and dedicated control of next events, setup internal flags, call and provide process call backs at significant milestones of the flow);
- HW setup necessary to keep and follow the flow (clearing interrupt event flags, enable disable associated HW events);
- return from interrupt service (restore context).

Bit-banging can be significantly faster but it is not easy task to keep any proper timing when this control is applied. It usually carries a software overhead consuming CPU cycles that they could be utilized for the other processes otherwise. This may have a noticeable effect on system responsiveness to other events, and in a hard real-time system, may significantly impact the system’s ability to meet real-time deadlines.

If the bit-banged interface is not to have a detrimental effect on real-time performance, then it must be performed with a low priority and then it can’t be deterministic in terms of data throughput and latency.

A specific problem could happen due to code alignment when the same code placed and executed from different starting addresses. This can produce different timing of provided signals. It is caused by different pipe line mechanism when code is fetched from the memory in advance or flushed out from pipeline. User has to respect setting of instruction prefetch and instruction and data cashing.

The most CPU efficient transfer is achieved by using a hardware interface and DMA transfer to minimize the software overhead and cpu load. Bit-banging is at the opposite extreme of that. It is neither portable nor flexible enough to handle all peripherals’ instances. The produced signals normally have more jitter or glitches, especially if the controller is also executing other tasks. That is why such communicating should be avoided in real time critical environment especially.

Using specific macros and conditional code can increase code visibility significantly and bring a sure level of flexibility even at this low level of programing (see the code examples).

Combination of bit-banging and interrupts can give a good fairly autonomous and interrupt driven bit-banging code, only at the expense of using a dedicated timer. But in general, it is very hard to do reliable bit-banging at high speeds in a multitasking environment.

2.4 Peripherals specific aspects of SW emulation

2.4.1 SPI

SW emulation of SPI is the simplest and easily adaptable as it is synchronous and no specific requirements limit the rates, timing of signals or physical interface characteristic.

Data transmission and reception is provided contemporary on two separated data unidirectional lines (MOSI, MISO) synchronized by common clock signal line provided by master. No addressing or acknowledgment control is implemented.

Concerning SCK clock signal, the user has to respect fixed clock phase and polarity between communicating nodes, even if different settings can be used for a group of nodes within a single net if their communication is held separated. For slave select signal, negative polarity is commonly expected. SS signal is often not implemented when simple communication is done between a single pair of nodes with fixed master and slave roles. Multi-master environment can be supported as well but there is no specific arbitration except SS signal level. SPI node can provide a transaction when its SS input stays inactive else multi-master conflict is recognized. In spite of full duplex communication is considered as standard case, many solutions use simplified simplex communication (SPI node can be transmitter or receiver only).

Clock signal can be continuous or not, no clock signal stretching is supported. This puts limits especially on slave, because any transaction can start or continue regardless if any data is ready or accepted on slave side. As a consequence, slave can easily enter in data underrun or data overrun conditions.

The number of bits to be transacted is fixed between nodes but arbitrarily configurable.

Optional API can integrate single data buffering or more complex FIFO structures for both transmitter and receiver.

2.4.2 UART

SW emulation of UART channel requires proper timing control as the data flow is asynchronous. Data transmission and reception are two quite independent and unsynchronized processes provided on two separated unidirectional data lines (Tx and Rx) driven exclusively by the transmitter.

Usually, any synchronization or timing issue should prevent timing errors larger than 30% within every transacted bit rate interval. When 10 bits are transacted within a character frame, the internal clock difference ratio between transmitter and receiver should not exceed 3% to handle the last transacted bit correctly. The defined rate and bit protocol are configurable.

Physical interface is mostly defined by additional communication standards like RS232/422/485.

Modem operation capability can be supported by additional status signals like CTS/RTS. No addressing or acknowledgment control is implemented.

Character framing is partially configurable, NRZ format is used. The frame is started by single start bit followed by 5-9 data bits with optional parity bit included and closed by 1-2 stop bits. If data line is held at idle longer than a character time a break condition can be detected. Bit sampling is usually provided in the middle of the bit interval and can include few samples of the same bit. It can raise noise, frame (stop) or parity framing errors. Receiver can face overrun error, too.

Optional API can integrate single data buffering or more complex FIFO structures for both transmitter and receiver.

2.4.3 I2C

SW emulation of I2C channel is the most complex one as wide range of aspects has to be considered.

A specific timing is required in dependency of selected mode, even if a synchronous protocol is used. This applies especially for Standard mode (for which rates are slower, up to 100 kbit/s), where duration of low and high period of SCL clock signal and margin between data SDA and clock SCL signals change during Start and Stop conditions have to be kept at a minimum value of about 5 µs.

The output stages of the bidirectional SDA and SCL signals should have an open-drain or open-collector to perform the wired-AND function. The standard allows to connect sources with different supply levels.

The protocol includes 10-bits or 7-bits addressing read and write modes, bus arbitration supporting multi master communication, data acknowledgment, clock stretching and clock synchronization procedures. The emulation task can be dramatically simplified when the application doesn’t need to support some of these requirements.

If API is applied, it is based mostly on more complex interface which collects all the necessary information to handle the protocol, data transfer and raised events. Misplaced start or stop condition (bus error), acknowledge failure and arbitration loss (at multi master system) are the communication errors usually signaled by the API. A slave receiver can face a data overrun when no clock stretching is applied, too.

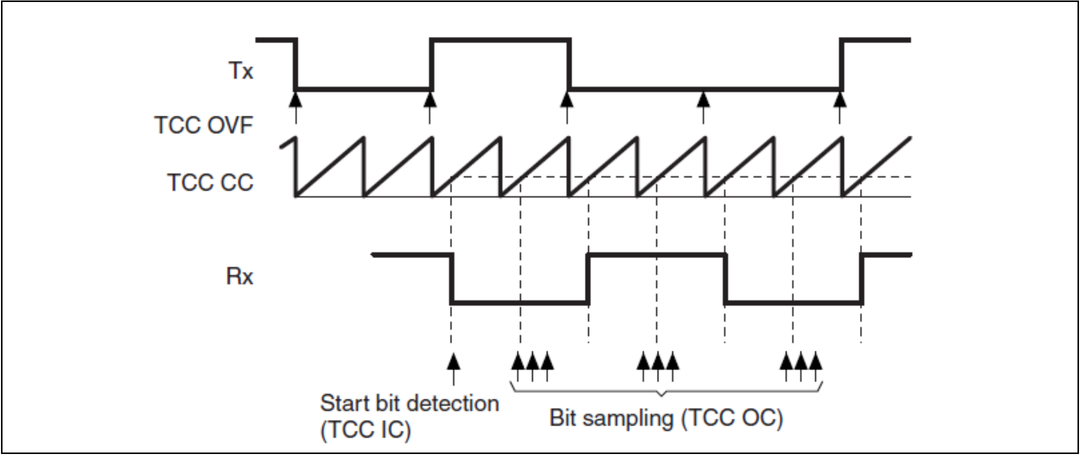
2.5 Examples

2.5.1 UART duplex channel based on combination of HW and SW control of GPIOs

This example shows how to control both transmit and receive processes by software, based on HW events.

Overflow and IO capture events of a single Capture & Compare timer is used to control timing of emulation of UART input sampling and output handling of the Rx and Tx signals. The associated input capture pin is dedicated to receive data. Any general-purpose output pin or associated output compare pin can be used for data transmission. The timer overflow period is set to the channel half bit rate by setting the timer auto reload register. *Figure 3* illustrates the timing diagram.

Figure 3. UART duplex channel based on timer capture events



Transmission process control is based on regular interrupts from the timer overflow events. After data for transmission is stored into dedicated variable (transmit buffer simulated by SW), an internal transmission request appears and the nearest overflow interrupt is used to start a new transmission (maximum latency for transmission start could be half a bit period). This is because the same timer is used for control both transmit and receive processes not synchronized normally. Every even timer overflow event interrupt controls the consecutive edge changes of the transmitted signal on the dedicated output pin until the end of the frame transmission. Odd overflow interrupts are discarded.

The receive process uses both input capture and output compare feature of the same timer and its dedicated pin. Initially, the input capture is performed at the pin. After detecting the first falling edge, the value captured in the capture register is then used for compare purposes because the input pin functionality is switched to output compare mode (without affecting any output GPIO capability since the pin still stays in input mode). Due to the half-a-bit overflow period of the timer, the nearest output compare event points to the middle of the first receive bit (start bit). Sampling can be simulated by three consecutive reading of the input pin level at that moment and if, for each of them, a low level is detected, the correctly received start bit is evaluated. The receive process then continues by watching every next odd output compare event. The same three point sampling method can be performed with noise detection logic here and for all the other received data bits until the end of the current receive frame. All even compare interrupts are discarded. After the stop bits of the frame are sampled, the Rx pin is switched back to input capture mode and waits for the next frame start condition. The detection of noise while the stop bits are being sampled should cause a frame error. If a noise is detected during start bit, the receive process should be aborted and the Rx pin switched back to input capture mode while waiting for the next falling edge capture.

User can build an API interface upon this low level HW abstraction level. It can include the UART channel initialization, enable and disable Rx and Tx channels, read and write the data flow (e.g. control the data buffers) and check the transactions' status. Size of data, parity control number of stop bits or other control can be solved on pre-compilation level by conditional compilation to speed up the code when these features are not used.

2.5.2 SPI bit-banging sequence

This is an example of procedure controlling SPI master full duplex 8 bits data frame transaction (MSB first format). The procedure can be written based as a sequence of applied macros to achieve a maximum speed, even if it doesn't fully respect good common software practice requiring structured code organization (e.g. collect repeated parts at loops or subroutines).

```
/**
  * @brief SPI master RW bit-bang
  * @param unsigned char wb - data to transmit
  * @retval unsigned char rb – data received
  */
unsigned char SPI_master_8_bit_data_RW (unsigned char wb)
{
  unsigned char msk = 0x80;
  unsigned char rb = 0;
  do
  {
  #if CPHA=0
    miso_control(rb);
    mosi_control(wb);
    setup_dly();
    sck_odd_edge_control();
    half_bit_dly();
    sck_even_edge_control();
    setup_dly();
  #else /* CPHA=1 */
    sck_odd_edge_control();
    setup_dly();
    miso_control(rb);
    mosi_control(wb);
    setup_dly();
    sck_even_edge_control();
    half_bit_dly();
  #endif /* CPHA */
  }
  while ((msk>>=1) != 0);
  return(rb)
}
```

The conditional code applied eliminates any residual testing inside the loop while sure level of universality and readability is still kept. If conditional compilation is not acceptable a set of separated procedures dedicated for concrete combination of setting provides faster code than single common procedure with integrated logic solving the variability of configuration inside.

All the SCK, MOSI and MOSI control signals should be defined like macros to insert either inline or no code if not used. The main effort is to prevent any function call at this section of code if the communication rate constraint is critical. If necessary, and if master is running too fast, macros defining SCK delays can be integrated to respect set up times for inputs and outputs or clock signal duty shape. They can be left mostly at zero when duration of instruction cycles' execution providing the bit-bang logic is comparable or even longer than minimum intervals required for data setup and hold times.

Related macros can be defined while using condition compilation like in the following:

```
#define spi_set_mosi() { SPI_DR |= (1 << SPI_MOSI_PIN); }
#define spi_clear_mosi() { SPI_DR &= ~(1 << SPI_MOSI_PIN); }
#define spi_get_miso()  (SPI_DR & (1 << SPI_MISO_PIN))

#if CPOL=0
  #define sck_odd_edge_control() { SPI_DR |= (1 << SPI_SCK_PIN); }
  #define sck_even_edge_control() { SPI_DR &= ~(1 << SPI_SCK_PIN); }
#else /* CPOL=1 */
  #define sck_odd_edge_control() { SPI_DR &= ~ (1 << SPI_SCK_PIN); }
  #define sck_even_edge_control() { SPI_DR |= (1 << SPI_SCK_PIN); }
#endif /* CPOL */

#ifdef SPI_MOSI
  #define mosi_control(b) { (b & msk) ?  spi_set_mosi() : spi_clear_mosi();
}
#else
  #define mosi_control(b) {} /* no code is provided */
#endif

#ifdef SPI_MISO
  #define miso_control(b) { if( spi_get_miso() != 0)  b |= msk;  }
#else
  #define miso_control(b) {} /* no code is provided */
#endif
```

```
#define setup_dly() {};/* no code is provided */
#define half_bit_dly() {};
```

2.5.3 I2C master bit-banging sequence

This example is related to procedures controlling I2C master 8 bits data frame read and write transactions.

```
#define i2c_bus_init() { I2C_DR |= ((1 << I2C_SDA_PIN) | (1 <<
I2C_SCL_PIN)); }
#define i2c_set_sda() { I2C_DR |= (1 << I2C_SDA_PIN); }
#define i2c_clear_sda() { I2C_DR &= ~(1 << I2C_SDA_PIN); }
#define i2c_get_sda()  (I2C_DR & I2C_SDA_PIN)
#define i2c_set_scl() { I2C_DR |= (1 << I2C_SCL_PIN); }
#define i2c_clear_scl() { I2C_DR &= ~(1 << I2C_SCL_PIN); }

#define sda_wr_control(b) { (b & msk) ?  i2c_set_sda() : i2c_clear_sda(); }
#define sda_rd_control(b) { I f(i2c_get_sda() != 0)  b |= msk;  }

/**
  * @brief I2C master write 8-bit data bit-bang
  * @param unsigned char b - data to transmit
  * @retval unsigned char ack - acknowledgement received
  */
void I2C_master_write (unsigned char b)
{
  unsigned char msk = 0x80;
  unsigned char ack;
  do
  {
    sda_wr_control(b);
    setup_dly();
    i2c_set_scl();
    half_bit_dly();
    i2c_clear_scl();
    setup_dly();
  }
  while ((msk>>=1) != 0);

  i2c_set_sda();/* ACK slot checking */
  i2c_set_scl();
  half_bit_dly();
  ack = i2c_get_sda();
  i2c_clear_scl();
  return (ack);
}
```

```
/**
  * @brief I2C master read 8-bit bit-bang
  * @param unsigned char ack - acknowledgement control
  * @retval unsigned char b - data received
  */
unsigned char I2C_master_read (unsigned char ack)
{
  unsigned char msk = 0x80;
  unsigned char b = 0;
  do
  {
    i2c_set_scl();
    half_bit_dly();
    sda_rd_control(b);
    i2c_clear_scl();
    half_bit_dly();
  }
  while ((msk>>=1) != 0);

  if(ack != 0)
  {
    i2c_clear_sda();/* ACK slot control */
  }
  setup_dly();
  i2c_set_scl();
  half_bit_dly();
  i2c_clear_scl();
  half_bit_dly();
  return (b);
}

/**
  * @brief I2C start
  * @param none
  * @retval none
  */
void I2C_Start_condition(void)
{
  i2c_bus_init();
  half_bit_dly();
  i2c_clear_sda();
  half_bit_dly();
  i2c_clear_scl()
  half_bit_dly();
}

/**
  * @brief I2C stop
  * @param none
  * @retval none
  */
void I2C_Stop_condition(void)
{
  i2c_clear_sda();
  i2c_clear_scl()
  half_bit_dly();
  i2c_set_scl();
  half_bit_dly();
  i2c_set_sda()
  half_bit_dly();
}
```

3 Conclusion

An effective emulation of a communication peripheral is feasible in principle with STM32 microcontrollers using methods described in this document.

User can benefit from this additional feature, provided he respects the limitations and considers a reasonable performance bandwidth. The task can be significantly simplified if the user identifies channels suitable for such emulation, and drops all not strictly needed requirements.

Any universality here should be carefully considered, to prevent the solution to become too complex or heavy from performance point of view.

4 Revision history

Table 1. Document revision history

Date	Revision	Changes
05-Mar-2015	1	Initial release